TP de Réseaux – RICM 4
Documentation d'aide à la programmation en langage C

Pascal Sicard

30 novembre 2011

Table des matières

1 Organisation du code source **3**

2 Compilation **4**

2.1 Utilisation de bibliothèques standard 4

3 Aide à la compilation : le `Makefile` **5**

4 Les sockets **6**

4.1 La fonction `socket` 6

4.2 La fonction `gethostbyname` 6

4.3 La fonction `getservbyname` 6

4.4 La fonction `inet_ntoa` - Obsolète remplacée par `inet_ntop` 7

4.5 La fonction `inet_ntop` 7

4.6 La fonction `inet_pton` 7

4.7 La fonction `gethostbyaddr` 7

4.8 Conversion Little endian /Big endian 8

4.9 La structure `sockaddr` 8

4.10 La fonction `getsockname` 9

4.11 La fonction `getpeername` 10

4.12 Les options des sockets 10

4.13 La fonction `connect` . . . 10
4.14 La fonction `close` . . . 10
4.15 La fonction `bind` . . . 11
4.16 La fonction `listen` . . . 11
4.17 La fonction `accept` . . . 11
4.18 Les fonctions `read` et `write` . . . 11
4.19 Les fonctions `sendto` et `recvfrom` . . . 12
4.20 La fonction `select` . . . 12
4.21 Canevas des algorithmes standards . . . 13

5 Les descripteurs de fichiers **13**
5.1 Les primitives associés aux descripteurs de fichier . . . 14
5.1.1 L'appel système `open()` . . . 14
5.1.2 L'appel système `read()` . . . 16
5.1.3 L'appel système `write()` . . . 16
5.1.4 L'appel système `close()` . . . 16
5.1.5 Les descripteurs de fichier particuliers . . . 16

6 Les fonctions d'entrées / sorties de la bibliothèque standard **16**
6.1 Les descripteurs de fichier particuliers . . . 17
6.2 La fonction `FILE * fopen(char *path, char *mode)` . . . 17
6.3 La fonction `int fclose(FILE *stream)` . . . 17
6.4 La fonction `int fread(char *buffer, int TailleBloc, int NbBloc, FILE *stream)` . . . 17
6.5 La fonction `int fwrite(char *buffer, int TailleBloc, int NbBloc, FILE *stream)` . . . 18
6.6 La fonction `int feof(FILE *stream)` . . . 18
6.7 La fonction `int ferror(FILE *stream)` . . . 18
6.8 La fonction `int fprintf(FILE *stream, const char *format, ...)` 18
6.9 La fonction `int fscanf(FILE *stream, const char *format, ...)` 19
6.10 La fonction `char * fgets(char *str, int size, FILE *stream)` . 19
6.11 La fonction `int getc(FILE *stream)` . . . 19

6.12 La fonction `int putc(int c, FILE *stream)` 19
6.13 La fonction `int fclose(FILE *stream)` 20
6.14 Les fonctions `int sprintf(char *str, const char *format, ...)` et `int sscanf(const char *str, const char *format, ...)` 20

7 Les Processus **20**
7.1 Les fonctions d'identification des processus 20
7.2 La création de processus 21
7.3 L'appel système wait 21
7.4 Les signaux 22
7.5 Liste et signification des différents signaux 23
7.6 Envoi d'un signal 23
7.7 Réception d'un signal 24

8 Autres primitives pouvant servir... **25**
8.1 La fonction `system()` 25
8.2 La fonction `popen()` 25

1 Organisation du code source

1. Le manuel en ligne
   La commande `man` permet de récupérer des informations sur les commandes, les fonctions des bibliothèques C ...
   Exemple : `man man` : donne le mode d'emploi de la commande `man`.
   L'option `-k` permet de chercher dans le manuel par mots clés. Exemple : `man -k stream`
2. Les fichiers sources
   - Fichiers.c (sources en C) :
     - Listes d'inclusions de fichiers.h
     - Listes de constantes
       Exemple :
       ```
       #define TRUC nimportequoi
       ```
       Attention c'est ensuite un « copie/collé » des chaînes de caractères (qui est effectué par le compilateur) dans la suite du fichier.
     - Listes de variables globales au fichier source
     - Listes de procédure/fonctions avec variables locales
     - Une procédure main particulière : programme principal
   - Fichiers.h (associés à un fichier de même nom suffixé *.c*)
     Permet d'inclure les déclarations de procédure/fonctions et constantes utilisés

dans un fichier.c pour faire de la compilation « séparée ». Une inclusion est équivalent à une recopie tel quel du fichier *.h* au moment de la compilation. Possibilité aussi de définitions de constantes.

3. Exemple :

```
proc1.h :
/* définition d'une constante */
#define CLIENT 3
/* entete de la procedure proc1*/
void proc1(int a) ;

proc1.c :
#include " proc1.h " /* ici proc1.h doit être dans le même répertoire */
void proc1(int a)
{/*corps de la procédure 1 ; utilisation de constante*/}

proc2.c
#include " proc1.h "
void proc2 (int b)
{/*corps de la procedure2 ; utilisation de proc1 et constante*/}
```

2 Compilation

- Un seul fichier c : `gcc fichier.c -o fichierexecutable`
- Plusieurs fichiers c : `gcc -c fichier1.c` : génération du code intermédiaire dans fichier1.o
  Puis `gcc fichier1.o fichier2.o -o fichierexecutable` : édition de lien entre les fichiers objets

2.1 Utilisation de bibliothèques standard

Inclusion dans fichier *.c* du fichier *.h* correspondant à la bibliothèque Exemple :

```
#include <socket.h>
 /* <.h> contrairement aux " .h " permet au compilateur de retrouvé
la bibliothèque (voir directive de compilation ci dessous)*/
```

Les options de compilation sont variables suivant les systèmes d'exploitation. Exemple : gcc fichier.c -o executable -lsocket

où socket est le nom de la bibliothèque dynamique (fichier : libsocket.a)

3 Aide à la compilation : le `Makefile`

Cet outil permet par la simple commande `make` de lancer les compilations des fichiers qui ont été modifiés depuis la dernière compilation.

Exemple de fichier makefile :

```
#commentaires
#définitions des fichiers objets comme une définition d'une constante en C
#(copie /collé dans la suite du fichier)

OBJ1 = fon.o client.o
OBJ2 = fon.o serveur.o

EXEC = ${OBJ1} client ${OBJ2} serveur

#But du makefile : créer ces fichiers
all: ${EXEC}

#Pour chacun d'eux on donne : son nom suivi des fichiers dont il dépend
#puis la commande de compilation qui le créera

fon.o :  fon.h fon.c
gcc -c fon.c

client.o : fon.h client.c
gcc  -c  client.c

serveur.o : fon.h serveur.c
gcc  -c  serveur.c

client : ${OBJ1}
gcc ${OBJ1} -o client -lcurses -lsocket -lnsl -ltermcap

serveur : ${OBJ2}
gcc ${OBJ2} -o serveur -lcurses -lsocket -lnsl -ltermcap

#permet de détruire tous les fichiers voulu (fichiers créés et core éventuel)
#par la commande make clean
clean :
rm -f ${EXEC} core
```

On peut aussi faire *make serveur* pour ne compiler que le serveur. On peut lancer un makefile de nom différent par *make -f nomMakedifférent*

ATTENTION : il faut utiliser des tabulations en début de ligne et non des espaces.

4 Les sockets

4.1 La fonction `socket`

```
int socket (int domaine, int mode, int protocole)
```

Elle permet de créer une socket de type quelconque :

- Le paramètre `domaine` précise la famille de protocoles à utiliser pour les communications. Il y a deux familles principales :
  - PF_UNIX pour les communications locales n'utilisant pas le réseau ; Communication entre processus sur une même machine
  - AF_INET pour les communications réseaux utilisant le protocole IP.

  Nous n'étudierons par la suite que la famille AF_INET.
- Le paramètre `mode` précise le type de la socket à créer :
  - SOCK_STREAM pour une communication bidirectionnelle sûre. Dans ce cas, la socket utilisera le protocole TCP pour communiquer.
  - SOCK_DGRAM pour une communication sous forme de datagrammes. Dans ce cas, la socket utilise le protocole UDP pour communiquer.
  - SOCK_RAW pour pouvoir créer soi-même ses propres paquets IP ou les recevoir sans traitement de la part de la couche Transport.
- Le paramètre `protocole` précise le protocole utilisé (dépend bien sûr du champ mode) :
  - IPPROTO_UDP
  - IPPROTO_TCP
  - IPPROTO_ICMP
  - IPPROTO_RAW

La valeur renvoyée par la fonction `socket()` est -1 en cas d'erreur, sinon c'est un descripteur de fichier qu'on peut par la suite utiliser avec les autres fonctions de la librairie des Sockets.

4.2 La fonction `gethostbyname`

```
struct hostent *gethostbyname(char* nom)
```

Elle permet d'interroger le DNS de façon à obtenir l'adresse IP d'une machine à partir de son nom. Cette fonction prend en argument une chaîne de caractères (pointeur de type `char *`) contenant le nom de la machine et renvoie un pointeur vers une structure de type `hostent` (défini dans le fichier d'en-tête <netdb.h>).

Cette structure contient trois champs qui nous intéressent :

- `h_name` est le nom canonique de la machine (pointeur de type `char *`) ;
- `h_addr_list` est la liste des adresse IP de la machine (pointeur de type `char **`). Peut aussi être vu dans le cas d'une seule adresse par le champ `h_addr (char *)`.
- `h_length` est la taille de chacune des adresses stockées dans `h_addr_list` (ceci sert à en permettre la recopie sans faire de suppositions sur la taille des adresses).

4.3 La fonction `getservbyname`

```
struct servent *getservbyname(char* nom, char *protocole )
```

Un nom peut être associé à un numéro de port. Sous Unix ces associations se trouvent dans le fichier */etc/services*. Le protocole utilisé (TCP/UDP) apparait aussi dans ce fichier. Le fonction

`getservbyname` permet de récupérer un numéro de port à partir de son nom et du protocole utilisé. Elle retourne un pointeur vers une structure de type `servent` :

```
struct servent {
char *s_name ;          /* nom du service */
char **s_aliases ; /* liste des alias */
int s_port ; /* numéro de port */
char *s_proto ;          /* protocole utilisé */
}
```

4.4 La fonction `inet_ntoa` - Obsolète remplacée par `inet_ntop`

```
char *inet_ntoa (struct in_addr in)
```

`inet_ntoa` convertit l'adresse Internet de l'hôte donne dans l'ordre des octets du réseau en une chaîne de caractères dans la notation décimale pointée.

4.5 La fonction `inet_ntop`

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size) ;
```

Permet de convertir une adresse IP issue de la structure *sockadr_in* en chaine de caractère sous forme "décimal pointé".
Exemple :
char str[INET_ADDRSTRLEN] ;
inet_ntop(AF_INET, &(adr_serv.sin_addr), str, INET_ADDRSTRLEN) ;

4.6 La fonction `inet_pton`

*int inet_pton(int af, const char *src, void *dst) ;*

Permet de remplir la structure *sockadr_in* par une adresse donnée en "décimal pointé" (chaîne de caractère).
Exemple :
struct sockaddr_in adr_serv ;
inet_pton(AF_INET, "192.0.0.1", &(adr_serv.sin_addr)) ;

4.7 La fonction `gethostbyaddr`

```
struct hostent *gethostbyaddr(void* addr, int tailleaddr, int protocole)
```

Elle retourne un pointeur sur la structure `hostent` à partir de l'adresse sous forme d'entier long (retourné par `inet_addr` par exemple).

4.8 Conversion Little endian /Big endian

Les ports et les adresses IP doivent être fourni sous un format unique. En effet les microprocesseurs peuvent avoir différentes conventions de stockages en mémoire des entiers courts (16 bits) et des entiers longs (32 bits) :
- `big endian` tels que les Motorola : octet de poids fort à l'adresse la plus petite
- `little endian` tels que les Intel : octet de poids faible à l'adresse la plus petite

Pour les communications IP, le format `big endian` a été retenu.

Il faut donc éventuellement convertir le numéro de port du format natif de la machine vers le format réseau au moyen de la fonction `htons()` (*host to network short*) pour les machines à processeur `little endian`.Pour rendre les programmes portables sur des machines de différent type, on utilise cette fonction sur tous les types de machines (elle est également définie sur les machines à base de processeur `big endian`, mais elle ne fait rien).

L'adresse IP n'a pas besoin d'être convertie, lorsqu'elle est issue des primitives `gethostbyname` et `inet_aton`.

`unsigned short htons(unsigned short port)`
`unsigned long htonl (unsigned long adresse)` Idem pour la conversion dans l'autre sens avec `ntohs` et `ntohl` (réseau vers machine).

4.9 La structure `sockaddr`

Dans le cas des communications IP, le pointeur vers la structure de type `sockaddr` est en fait un pointeur vers une structure de type `sockaddr_in`, définie dans le fichier d'en-tête <netinet/in.h> (d'où la conversion de pointeur).

Cette structure contient trois champ utiles :
- `short sin_family` est la famille de protocoles utilisée, `AF_INET` pour IP ;
- `unsigned short sin_port` est le port à affecter à la structure ; (0 pour une allocation dynamique côté client)
- `unsigned long sin_addr.s_addr` est l'adresse IP à affecter à la structure.

Dans le cas de l'adresse locale d'un serveur ce sera la constante `IN_ADDR_ANY` pour pouvoir recevoir des demandes sur toutes les interfaces possibles du serveur.

La fonction `memset()` permet d'initialiser globalement la structure à zéro afin de supprimer toute valeur parasite.

Exemple :

```
...
   struct hostent       *p_info_serveur ; /* pointeur sur infos serveur  */
   struct p_info_service
   struct sockaddr_in   *p_adr_serv ;
   char* serveur ; /* nom du serveur soit DNS, soit en décimale pointée */
...
  /* Mise a zero de la structure d'adresse socket */
memset( (char *) p_adr_serv, 0, sizeof(*p_adr_serv) );

  /* Definition du domaine ( famille )  */
```

```
p_adr_serv->sin_family = AF_INET;

  /* ------ RENSEIGNE adresse IP ----------------------------------------*/
        if (serveur==INADDR_ANY) /* Cas d'un serveur */
     p_adr_serv->sin_addr.s_addr =htonl( INADDR_ANY);
  else
            {
            p_info_serveur = gethostbyname( serveur );
            if (p_info_serveur) /* Serveur reference dans /etc/hosts ou DNS */
{
memcpy ( (char *)&p_adr_serv->sin_addr,
 p_info_serveur->h_addr,p_info_serveur->h_length );
/*  ici pas besoin de htonl*/
}
                else
{
/* Adresse IP sous forme de decimale pointee */
/* sous forme de chaine de car */
p_adr_serv->sin_addr.s_addr = inet_addr(serveur) ;
/*  ici pas besoin de htonl*/
}
    }
  /* --------- RENSEIGNE PORT -----------------------------------------*/

p_info_service = getservbyname(service, protocole);

if ( p_info_service )         /* Service reference dans /etc/service */
{
 p_adr_serv->sin_port = htons ( ntohs(p_info_service->s_port)
+ DEFPORTBASE );
}
else
/* Service repere par un entier ( sous forme de chaine de car ) */
/* il faut en deduire le numero de port */
{
p_adr_serv->sin_port = htons( (u_short)atoi(service) );
}
 ...
```

4.10 La fonction `getsockname`

`int getsockname ( int socket, struct sockaddr_in *name, int *namelen )`
Cette procédure permet de récupérer l'adresse et le port (dans la structure `sockaddr_in`) locaux de la socket donnée en paramètre (utile en cas d'allocation dynamique du numéro de port). ATTENTION : le paramètre `namelen` est un paramètre « donnée/résultat ». La fonction retourne 1 en cas d' échec de la recherche.

4.11 La fonction `getpeername`

```
int getpeername ( int socket, struct sockaddr_in *name, int *namelen )
```

Cette procédure permet de récupérer l'adresse et le port (dans la structure `sockaddr_in`) du processus distant connecté à la socket donnée en paramètre. La fonction retourne 1 en cas d'échec de la recherche.

4.12 Les options des sockets

- `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)` permet de récupérer les valeurs des options
  - s : descripteur de socket
  - level : numéro de protocole (renvoyé par la fonction `getprotobyname`)
  - optname : nom de l'option
- `int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen)` permet de mettre à jour les options d'une socket
  Quelques options intéressantes(voir le man) Pour TCP et UDP :
  - SO_REUSEADDR enable/disable local address reuse
  - SO_OOBINLINE enable/disable reception of out-of-band data in band
  - SO_SNDBUF set buffer size for output
  - SO_RCVBUF set buffer size for input

Exemple :

```
int optval=5000 ;
setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (char *)&optval, sizeof(int)) ;
```

Possibilité de gérer avec ces options des sockets multicast (voir le man et les sources de socklab).

4.13 La fonction `connect`

```
int connect (int sock, struct sockaddr_in *paddr, int lg_struct)
```

Cette fonction prend en argument :

1. le descripteur de fichier d'une socket préalablement créée ;
2. un pointeur vers une structure de type `sockaddr_in` qui comporte le port et l'adresse de la socket distante ;
3. la taille de cette structure.

Elle retourne 1 si la connexion n'est pas possible 0 sinon.

4.14 La fonction `close`

```
int close(int sock)
```

La socket se ferme simplement au moyen de la fonction `close()`. On peut aussi utiliser la fonction `shutdown()`, qui permet une fermeture sélective de la socket (soit en lecture, soit en écriture).

4.15 La fonction `bind`

```
int bind(int sock, struct sockaddr_in *paddr, int lg_struct)
```

Cette fonction prend en argument :

1. le descripteur de fichier de la socket préalablement créée ;
2. un pointeur vers une structure de type `sockaddr_in` (port et adresse locaux) ;
3. la taille de cette structure.

4.16 La fonction `listen`

```
int listen(int sock, int nb_connexion)
```

Elle permet d'informer le système d'exploitation de la présence du nouveau serveur :

Cette fonction prend en argument :

1. le descripteur de fichier d'une socket préalablement créée ;
2. le nombre maximum de connexions pouvant être en attente (on utilise généralement la constante `SOMAXCONN` qui représente le nombre maximum de connexions en attente autorisé par le système d'exploitation).

4.17 La fonction `accept`

```
int accept(int sock, struct sockaddr_in *paddr, int *lg)
```

Elle permet d'attendre qu'un client se connecte au serveur :

Cette fonction prend trois arguments :

1. le numéro de descripteur de la socket ;
2. un pointeur vers une structure de type `sockaddr`, structure qui sera remplie avec les paramètres du client qui s'est connecté ;
3. un pointeur vers un entier, qui sera rempli avec la taille de la structure ci-dessus.

En cas de succès de la connexion, la fonction `accept()` renvoie un nouveau descripteur de fichier représentant la connexion avec le client. Le descripteur initial peut encore servir à de nouveaux clients pour se connecter au serveur, c'est pourquoi les serveurs appellent ensuite généralement la fonction `fork()` afin de créer un nouveau processus chargé du dialogue avec le client tandis que le processus initial continue à attendre des connexions.

Après un *fork()*, chaque processus ferme le descripteur qui lui est inutile.

4.18 Les fonctions `read` et `write`

```
int read (int sock, char *buffer, int nb_a_lire)
int write(int sock, char *buffer, int nb_a_ecrire)
```

Elles sont utilisées en mode TCP ou inter-processus.

`read` retourne le nombre d'octets effectivement lus. `write` retourne le nombre d'octets effectivement écrits. En cas d'erreur elles retournent la valeur `-1`.
En cas de fermeture de la socket par l'autre entité (réception du flag `fin` au niveau TCP), `read` retourne la valeur `0`.

4.19 Les fonctions `sendto` et `recvfrom`

`int sendto (int sock, char *buffer, int nb_a_ecrire, int flags, struct sockaddr_in *p_adr_distant, int taille_struct)`
`int recvfrom (int sock, char *buffer, int nb_a_lire, int flags, struct sockaddr_in *p_adr_distant, int taille_struct)`

Elles sont utilisées en mode UDP ou RAW `sendto` retourne le nombre d'octets effectivement écrits. `rcvfrom` retourne le nombre d'octets effectivement écrits.

`p_adr_distant` est un pointeur vers la structure qui comporte le port et l'adresse distants.

4.20 La fonction `select`

`int select (int maxfdpl, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`
Pour pouvoir écouter deux descripteurs à la fois et lire les données sur un descripteur quand elles sont disponibles, on utilise la fonction `select()`, qui est extrêmement pratique et qui peut s'utiliser avec tout type de descripteurs de fichier (socket, tuyaux, entrée standard...). Elle peut être utilisé pour mélanger des sockets et des entrées / sorties standard.

Exemple :

```
main ()
{
fd_set set, setbis ;
int idsock1, idsock2, maxsock ;
...
FD_ZERO(&set) ;
maxsock= getdtablesize() ;
FD_SET(idsock1, &set) ; /* ajout de idsock1 à l'ensemble set */
FD_SET(idsock2, &set) ; /* ajout de idsock2 à l'ensemble set */

bcopy ( (char*) &set, (char*) &setbis, sizeof(set)) ;
/* copie de l'ensemble set dans setbis */
for(... ;... ;...)
{
select (maxsock,&set, 0, 0, 0) ;
if (FD_ISSET(idsock1, &set))  /* Test si idsock1 appartient à l'ensemble set */
...
if (FD_ISSET(idsock2, &set))
bcopy ( (char*) &setbis, (char*) &set, sizeof(set)) ;
```

```
}
...
}
```

On commence par indiquer, au moyen d'une variable de type `fd_set`, quels descripteurs nous intéressent.

La fonction `FD_ZERO()` met la variable set à zéro, puis on indique son intérêt dans l'écoute des sockets grâce à la fonction `FD_SET()`. On appelle ensuite la fonction `select()` qui attendra que des données soient disponibles sur l'un de ces descripteurs :

La fonction `select()` prend cinq arguments :

1. Le plus grand numéro de descripteur à surveiller, plus un. Dans notre exemple, c'est `maxsock` mais cela pourrait etre (`max(idsocket1,idsocket2) + 1`).
2. Un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels on veut lire. Au retour de `select` cette liste contient les descripteurs prêts.
3. Un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels on veut écrire. N'étant pas intéressés par cette possibilité, nous avons utilisé un pointeur nul.
4. Un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels peuvent arriver des conditions exceptionnelles. N'étant pas intéressés par cette possibilité, nous avons utilisé un pointeur nul.
5. Un pointeur vers une structure de type `timeval` représentant une durée après laquelle la fonction `select` doit rendre la main si aucun descripteur n'est disponible. Dans ce cas, la valeur retournée par `select` est 0. N'étant pas intéressés par cette possibilité, nous avons utilisé un pointeur nul.

La fonction `select()` renvoie -1 en cas d'erreur, 0 au bout du temps spécifié par le cinquième paramètre et le nombre de descripteurs prêts sinon. L'ensemble des descripteurs contient après l'appel à select les descripteurs prêts à être lus.

La fonction `FD_ISSET` permet de déterminer si un descripteur est dans un ensemble.

4.21 Canevas des algorithmes standards

La figure 1 donne le canevas de l'algorithme du client utilisant le protocole TCP. La figure 2 donne le canevas de l'algorithme du serveur utilisant le protocole TCP et traitant plusieurs clients en parallèle.

5 Les descripteurs de fichiers

Cette section ne concerne que les appels systèmes associés à l'utilisation de fichiers. Les fonctions de la bibliothèque standard d'entrées / sorties ayant des fonctionnalités comparables sont décrites dans la section suivante. Ces "appels système" sont des opérations de « bas niveau », adaptées à certains types de manipulation de fichiers, comme par exemple des écritures brutales (i.e. sans formatage) de données stockées de façon contiguë en mémoire.

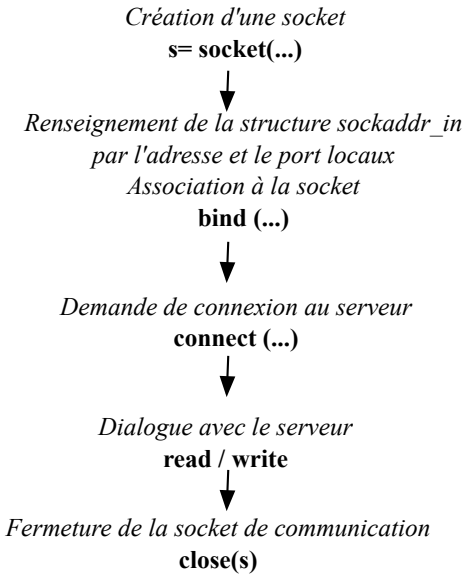

FIG. 1 – Canevas du client utilisant TCP

Le système d'exploitation tient à jour une table, appelée table des fichiers ouverts, où sont référencés tous les fichiers utilisés, c'est-à-dire tous les fichiers en train d'être manipulés par un processus (création, écriture, lecture).

Le mot fichier ne doit pas être compris ici au sens « fichier sur le disque dur », mais comme une entité pouvant contenir ou transmettre des données. Un descripteur de fichier peut aussi bien faire référence à un fichier du disque dur, à un terminal, à une connexion réseau ou à un lecteur de bande Magnétique. Un certain nombre d'opérations génériques sont définies sur les descripteurs de fichier, qui sont ensuite traduites par le système en fonction du périphérique auquel se rapporte ce descripteur. Ainsi, écrire une chaîne de caractères à l'écran ou dans un fichier se fera - pour l'utilisateur - de la même manière. D'autres opérations sont spécifiques au type de descripteur de fichier (socket par exemple).

5.1 Les primitives associés aux descripteurs de fichier

Les déclarations des appels système décrits dans cette section se trouvent dans les fichiers suivants : `/usr/include/unistd.h`
`/usr/include/sys/types.h`
`/usr/include/sys/stat.h`
`/usr/include/fcntl.h`

5.1.1 L'appel système `open()`

```
int open(const char *path, int flags, mode_t mode)
```

Il permet d'associer un descripteur de fichier à un fichier que l'on souhaite manipuler. En cas de succès, le système d'exploitation va créer une référence dans la table des fichiers et va indiquer

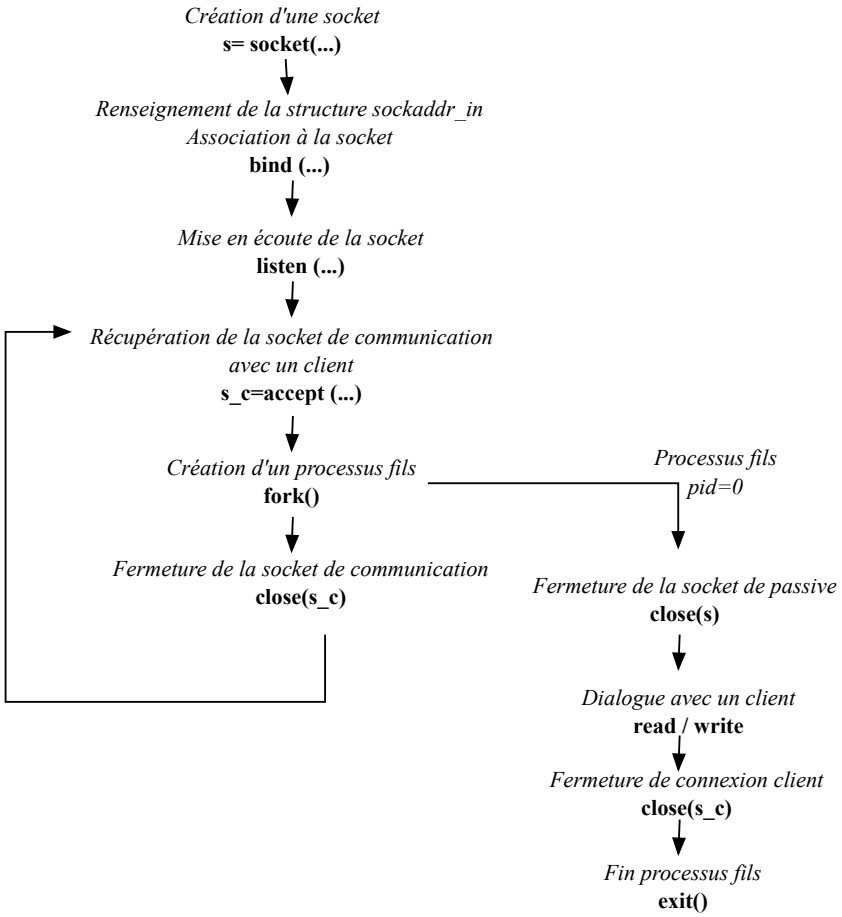

FIG. 2 – Canevas du serveur parallèle utilisant TCP

l'entier correspondant. Une fois référencé dans la table des fichiers, le fichier est dit « ouvert ». En cas d'erreur, par exemple lorsque le fichier désigné n'existe pas, open() retourne -1.

Le paramètre `path` est une chaîne de caractères donnant le chemin d'accès du fichier.

Le paramètre `flags` détermine de quelle façon le fichier va être ouvert, c'est-à-dire quels types d'opérations vont être appliquées à ce fichier :

- `O_RDONLY` pour un fichier ouvert en lecture seule ;
- `O_WRONLY` pour un fichier ouvert en écriture seule ;
- `O_RDWR` pour un fichier ouvert en lecture/écriture ;
- `O_APPEND` si le fichier existe, les données qui seront ajoutées seront placées à la fin du fichier et n'écraseront pas les données existantes ;
- `O_CREAT` crée le fichier s'il n'existe pas.

La valeur de flags peut aussi être une combinaison des valeurs ci-dessus (par la fonction « ou bit à bit » —). Par exemple, `O_RDONLY | O_WRONLY` qui doit être sensiblement équivalent à `O_RDWR`.

Le paramètre mode n'est utilisé que si le drapeau `O_CREAT` est présent. Dans ce cas-là, mode indique les permissions du fichier créé : les 3 chiffres représentent les permissions pour (de gauche à droite) l'utilisateur, le groupe et les autres (voir indications données par la commande *ls -l*).

5.1.2 L'appel système `read()`

```
int read(int fd, void *buf, size_t nbytes)
```

Le paramètre `fd` indique le descripteur de fichier concerné, c'est-à-dire le fichier dans lequels les données vont être lues.

Le paramètre `buf` désigne une zone mémoire dans laquelle les données lues vont être stockées. Cette zone mémoire doit être préalable allouée. Le paramètre `nbytes` indique le nombre d'octets (et non le nombre de données) que l'appel `read()` va essayer de lire (il se peut qu'il y ait moins de données que le nombre spécifié par `nbytes`). Le type `size_t` est en fait équivalent à un type int et a été défini dans un fichier d'en-tête par un appel à typedef.

La valeur retournée par `read()` est le nombre d'octets effectivement lus. À la fin du fichier (c'est-à-dire quand il n'y a plus de données disponibles), 0 est retourné. En cas d'erreur, -1 est retourné.

5.1.3 L'appel système `write()`

```
int write(int fd, void *buf, size_t nbytes)
```
Il s'utilise de la même façon que l'appel système `read()`.

La valeur retournée est le nombre d'octets effectivement écrits. En cas d'erreur, -1 est retourné.

5.1.4 L'appel système `close()`

```
int close(int fd)
```
Cet appel permet de fermer un fichier précédemment ouvert par l'appel système `open()`. En cas d'oubli, le système détruira de toute façon en fin de programme les ressources allouées pour ce fichier. Il est néanmoins préférable de fermer explicitement les fichiers qui ne sont plus utilisés.

5.1.5 Les descripteurs de fichier particuliers

Les descripteurs de fichiers 0, 1 et 2 sont spéciaux : ils représentent respectivement l'entrée standard, la sortie standard et la sortie d'erreur. Si le programme est lancé depuis un terminal de commande sans redirection, 0 est associé au clavier, 1 et 2 à l'écran.

6 Les fonctions d'entrées / sorties de la bibliothèque standard

Les fonctions d'entrées / sorties de la bibliothèque standard `stdio.h` ont les caractéristiques suivantes :

- Les accès aux fichiers sont asynchrones et « bufferisés ». Cela signifie que les lectures ou les écritures de données dans un fichier n'ont pas lieu au moment où elles sont demandées, mais qu'elles sont déclenchées ultérieurement. En fait, les fonctions de la bibliothèque standard

utilisent des zones de mémoire tampon (des buffers en anglais) pour éviter de faire trop souvent appel au système et elles vident ces tampons soit lorsque cela leur est demandé explicitement, soit lorsqu'ils sont pleins.
- Les accès aux fichiers sont formatés, ce qui signifie que les données lues ou écrites sont interprétées conformément à un type de données (int, char, etc.).

Ces fonctions sont particulièrement indiquées pour traiter un flot de données de type texte.

6.1 Les descripteurs de fichier particuliers

Trois descripteurs de fichiers particuliers sont prédéfinis :
- `stdin` qui correspond au descripteur de fichier entier 0, c'est-à-dire à l'entrée standard ;
- `stdout` qui correspond au descripteur de fichier entier 1, c'est-à-dire à la sortie standard ;
- `stderr` qui correspond au descripteur de fichier entier 2, c'est-à-dire à la sortie d'erreur standard.

Les fonctions `scanf()` et `printf()` travaillent respectivement sur `stdin` et `stdout`.

6.2 La fonction `FILE * fopen(char *path, char *mode)`

Elle ouvre le fichier indiqué par la chaîne de caractères `path` et retourne soit un pointeur sur une structure de type `FILE` décrivant le fichier, soit la valeur symbolique `NULL` si une erreur est survenue.
La chaine `mode` indique le type d'ouverture :
- "r" (pour read) ouvre le fichier en lecture seule ;
- "r+" ouvre le fichier en lecture/écriture ;
- "w" (pour write) crée le fichier s'il n'existe pas ou le tronque à 0 et l'ouvre en écriture ;
- "w+" comme "w" mais ouvre le fichier en lecture/écriture ;
- "a" (pour append) crée le fichier s'il n'existe pas et l'ouvre en écriture, en positionnant le descripteur à la fin du fichier ;
- "a+" comme "a" mais ouvre le fichier en lecture/écriture ;

Exemple : `f = fopen("/tmp/test", "w ");`

6.3 La fonction `int fclose(FILE *stream)`

Elle permet de fermer un fichier préalablement ouvert à l'aide de `fopen`. Elle retourne 0 si la fermeture s'est bien passée.

6.4 La fonction `int fread(char *buffer, int TailleBloc, int NbBloc, FILE *stream)`

`fread` permet de lire dans un fichier `NbBloc` de taille `TailleBloc` caractères. Les caractères sont copiés dans le buffer qui doit être préalablement alloué. La valeur retournée est le nombre de bloc effectivement lu. Si la valeur retournée est 0, soit la fin du fichier a été rencontrée, soit une erreur est survenue. On utilisera une taille de bloc à 1 pour une lecture de caractères.

6.5 La fonction `int fwrite(char *buffer, int TailleBloc, int NbBloc, FILE *stream)`

`fwrite` fonctionne de la même manière que `fread` mais en écriture. La valeur retournée est égale au nombre de blocs effectivement écrits. Si elle est différent de `NbBloc`, il y a eu une erreur.

6.6 La fonction `int feof(FILE *stream)`

Cette fonction retourne 0 si le fichier décrit par `stream` contient encore des données à lire, une valeur différente de 0 sinon.

6.7 La fonction `int ferror(FILE *stream)`

Cette fonction retourne une valeur différente de 0 si une erreur est survenue lors de la dernière opération sur le fichier `stream`. Exemple :

```
if (ferror(f) != 0)  {
     printf("erreur en écriture sur le fichier\n");
     exit(EXIT_FAILURE);
 }
```

6.8 La fonction `int fprintf(FILE *stream, const char *format, ...)`

Elle écrit dans le fichier décrit par `stream` les données spécifiés par les paramètres suivants. La chaîne de caractère `format` peut contenir des données à écrire ainsi que la spécification d'un format d'écriture à appliquer.

```
%d : entier
%c : caractère
%s : chaîne de caractères
%f : flottant
```

Cette fonction s'utilise de la même manière que la fonction `printf()`. La valeur retournée est le nombre de caractères écrits. Exemple :

```
fprintf(f, "la valeur de i est : %d, la valeur de c est : %c\n", i, car)};
```

6.9 La fonction `int fscanf(FILE *stream, const char *format, ...)`

Cette fonction lit le fichier décrit par `stream`, selon le format spécifié par la chaîne `format` et stocke les valeurs correspondantes aux adresses spécifiées par les paramètres suivants. Attention, les adresse spécifiées doivent être valides, c'est-à-dire qu'elles doivent correspondre à des zones mémoires préalablement allouées.

La valeur retournée est soit le nombre de conversions effectuées, soit la valeur symbolique `EOF` si aucune conversion n'a eu lieu. Exemple :

```
fscanf(f, "%d %d", &i,&j); /* un blanc doit séparé les entiers dans le fichier */}
```

6.10 La fonction `char * fgets(char *str, int size, FILE *stream)`

Cette fonction lit des caractères depuis le fichier `stream` et les stocke dans la chaîne de caractères `str`. La lecture s'arrête soit après `size` caractères, soit lorsqu'un caractère de fin de ligne est rencontré, soit lorsque la fin du fichier est atteinte. La fonction `fgets()` retourne soit le pointeur sur `str`, soit `NULL` si la fin du fichier est atteinte ou si une erreur est survenue.

Comme pour `fscanf()`, les fonctions `ifeof` et `ferror` peuvent être utilisées pour savoir si la fin du fichier a été atteinte ou si une erreur est survenue.

Exemple :

```
char buf[255];
...
fgets(buf, 255, f);
...
```

6.11 La fonction `int getc(FILE *stream)`

Cette fonction lit un caractère depuis le fichier `stream` et le retourne sous forme d'entier. `int getchar()` fait la même chose avec l'entrée standard (clavier) `stdin`.

6.12 La fonction `int putc(int c, FILE *stream)`

Cette fonction écrit le caractère `c` dans le fichier `stream` .
`int putchar (int c)` fait la même chose avec `stdout`.

6.13 La fonction `int fclose(FILE *stream)`

Cette fonction indique au système que le fichier `stream` ne sera plus utilisé et que les ressources associées peuvent être libérées. La fonction retourne `EOF` en cas d'erreur ou 0 sinon.

6.14 Les fonctions `int sprintf(char *str, const char *format, ...)` et `int sscanf(const char *str, const char *format, ...)`

Elles fonctionnent de la même manière que les fonctions `fprintf()` et `fscanf()`, mais prennent ou stockent leurs données dans une chaîne de caractères et non dans un fichier.

7 Les Processus

Les ordinateurs dotés de systèmes d'exploitation modernes peuvent exécuter « en même temps » plusieurs programmes pour plusieurs utilisateurs différents : ces machines sont dites multi-tâches et multi-utilisateurs. Un processus (ou process en anglais) est un ensemble d'instructions se déroulant séquentiellement sur le processeur (par exemple, un de vos programmes en C, le shell qui interprète les commandes entrées au clavier, ou le programme X qui gère l'écran de votre machine).

Sous Unix, la commande *ps* permet de voir la liste des processus existant sur une machine : *ps -xu* donne la liste des processus que vous avez lancé sur la machine, avec un certain nombre d'informations sur ces processus. En particulier, la première colonne donne le nom de l'utilisateur ayant lancé le processus, et la dernière le contenu du tableau `argv` du processus. *ps -aux* donne la liste de tout les processus lancés sur la machine.

Chaque processus, à sa création, se voit attribuer un numéro d'identification (le *pid*). C'est ce numéro qui est utilisé ensuite pour désigner un processus. Ce numéro se trouve dans la deuxième colonne de la sortie de la commande *ps -axu*.

Un processus ne peut être créé qu'à partir d'un autre processus (sauf le premier, *init*, qui est créé par le système au démarrage de la machine). Chaque processus a donc un ou plusieurs fils, et un père, ce qui crée une structure arborescente. À noter que certains systèmes d'exploitation peuvent utiliser des processus pour leur gestion interne, dans ce cas le pid du processus *init* (le premier processus en mode utilisateur) sera supérieur à 1.

7.1 Les fonctions d'identification des processus

Ces fonctions sont au nombre de deux :

- `pid_t getpid(void)` Cette fonction retourne le pid du processus.
- `pid_t getppid(void)` Cette fonction retourne le *pid* du processus père. Si le processus père n'existe plus (parce qu'il s'est terminé avant le processus fils, par exemple),

la valeur retournée est celle qui correspond au processus *init* (en général 1), ancêtre de tous les autres et qui ne se termine jamais.

7.2 La création de processus

La bibliothèque C propose plusieurs fonctions pour créer des processus avec des interfaces plus ou moins perfectionnées. Cependant toutes ces fonctions utilisent l'appel système `fork()` qui est la seule et unique façon de demander au système d'exploitation de créer un nouveau processus.

`pid_t fork(void)`
Cette fonction est un "appel système" qui crée un nouveau processus. La valeur retournée est le *pid* du fils pour le processus père, et 0 pour le processus fils. La valeur 1 est retournée en cas d'erreur.

`fork()` se contente de dupliquer un processus en mémoire, c'est-à-dire que la zone mémoire du processus père est recopiée dans la zone mémoire du processus fils. On obtient alors deux processus identiques, déroulant le même code en concurrence. Ces deux processus ont les mêmes descripteurs de fichiers, de socket et tuyaux. Seule la valeur retournée par le `fork()` est différente suivant dans quel processus on se trouve. La valeur retournée par `fork` est donc utilisée pour déterminer si on est le processus père ou le processus fils et permet d'agir en conséquence. Attention, les variables ne sont par contre pas partagées.

L'exemple suivant montre l'effet de fork() :

```
#include <unistd.h>
int main(int argc, char **argv)
{   int i;
    printf("je suis avant le fork\n");
    i = fork();
    printf("je suis apres le fork, il a retourne %d\n", i);
}
```

Le résultat de l'exécution est :

```
je suis avant le fork
je suis apres le fork, il a retourne 3214
je suis apres le fork, il a retourne 0
```

Lors de l'exécution, l'ordre dans lequel le fils et le père affichent leurs informations n'est pas toujours le même. Cela est dû à l'ordonnancement des processus par le système d'exploitation qui ne s'effectue pas toujours dans le même ordre.

7.3 L'appel système wait

Il est souvent très pratique de pouvoir attendre la fin de l'exécution des processus fils avant de continuer l'exécution du processus père (afin d'éviter que celui-ci se termine avant ses fils, par

exemple).

La fonction `int wait(int *status)` permet de suspendre l'exécution du père jusqu'à ce que l'exécution d'un des fils soit terminée. La valeur retournée est le pid du processus qui vient de se terminer ou -1 en cas d'erreur. Si le pointeur status est différent de NULL, les données retournées contiennent des informations sur la manière dont ce processus s'est terminé (comme par exemple la valeur passée à *exit()*).

Dans l'exemple suivant, nous complétons le programme décrit précédemment en utilisant `wait()` : le père attend alors que le fils soit terminé pour afficher les informations sur le `fork()` et ainsi s'affranchir de l'ordonnancement aléatoire des tâches.

```
int main(int argc, char **argv)
{   int i,j;
    printf("je suis avant le fork\n");
    i = fork();
    if (i != 0) { /* i != 0 seulement pour le pere */
        j = wait(NULL);
        printf("wait a retourne %d\n", j);
    }
    printf("je suis apres le fork, il a retourne %d\n", i);
}
```

L'exécution donne :

```
je suis avant le fork
je suis apres le fork, il a retourne 0
wait a retourne 473
je suis apres le fork, il a retourne 473
```

Notons que cette fois-ci, l'ordre dans lequel les informations s'affichent est toujours le même.

7.4 Les signaux

Les signaux constituent la forme la plus simple de communication entre processus.

Un signal est une information atomique envoyée à un processus ou à un groupe de processus par le système d'exploitation ou par un autre processus. Lorsqu'un processus reçoit un signal, le système d'exploitation l'informe : « tu as reçu tel signal », sans plus. Un signal ne transporte donc aucune autre information utile.

Au niveau du microprocesseur, un signal correspond à une interruption logicielle.

Lorsqu'il reçoit un signal, un processus peut réagir de trois façons :
- Il est immédiatement dérouté vers une fonction spécifique, qui réagit au signal (en modifiant la valeur de certaines variables ou en effectuant certaines actions, par exemple). Une fois cette fonction terminée, on reprend le cours normal de l'exécution du programme, comme si rien ne s'était passé.
- Le signal est tout simplement ignoré.

• Le signal provoque l'arrêt du processus (avec ou sans génération d'un fichier core).

Lorsqu'un processus reçoit un signal pour lequel il n'a pas indiqué de fonction de traitement, le système d'exploitation adopte une réaction par défaut qui varie suivant les signaux :

• soit il ignore le signal ;

• soit il termine le processus (avec ou sans génération d'un fichier core).

Vous avez certainement toutes et tous déjà utilisé des signaux, consciemment, en tapant `Control-C` ou en employant la commande `kill`, ou inconsciemment, lorsqu'un de vos programmes a affiché : "segmentation fault (core dumped)"

7.5 Liste et signification des différents signaux

La liste des signaux dépend du type d'UNIX. La norme POSIX.1 en spécifie un certain nombre, parmi les plus répandus. On peut néanmoins dégager un grand nombre de signaux communs à toutes les versions d'UNIX :

SIGHUP rupture de ligne téléphonique. Du temps où certains terminaux étaient reliés par ligne téléphonique à un ordinateur distant, ce signal était envoyé aux processus en cours d'exécution sur l'ordinateur lorsque la liaison vers le terminal était coupée. Ce signal est maintenant utilisé pour demander à des démons (processus lancés au démarrage du système et tournant en tâche de fond) de relire leur fichier de configuration.

SIGINT interruption (c'est le signal qui est envoyé à un processus quand on tape Control-C au clavier).

SIGFPE erreur de calcul en virgule flottante, le plus souvent une division par zéro.

SIGKILL tue le processus.

SIGBUS erreur de bus.

SIGSEGV violation de segment, généralement à cause d'un pointeur nul.

SIGPIPE tentative d'écriture dans un tuyau qui n'a plus de lecteurs. (voir tuyaux)

SIGALRM Généré par la fonction alarm (int NbSeconde).

SIGTERM demande au processus de se terminer proprement.

SIGCHLD indique au processus père qu'un de ses fils vient de se terminer. A ignorer dans un processus père qui pourrait se finir avant ses fils . Sinon les processus fils peuvent devenir zombie.

SIGWINCH indique que la fenêtre dans lequel tourne un programme a changé de taille.

SIGUSR1 signal utilisateur 1.

SIGUSR2 signal utilisateur 2.

Il n'est pas possible de dérouter le programme vers une fonction de traitement sur réception du signal SIGKILL, celui-ci provoque toujours la fin du processus. Ceci permet à l'administrateur système de supprimer n'importe quel processus.

À chaque signal est associé un numéro. Les correspondances entre numéro et nom des signaux se trouvent généralement dans le fichier /usr/include/signal.h ou dans le fichier /usr/include/sys/signal.h suivant le système.

7.6 Envoi d'un signal

La fonction : `int kill (int pid,int signum)` permet d'envoyer un signal à un processus : Exemple : `kill(1664,SIGHUP)`

Ici, on envoie le signal SIGHUP au processus numéro 1664.

7.7 Réception d'un signal

L'interface actuelle de programmation des signaux (qui respecte la norme POSIX.1) repose sur la fonction `sigaction()`. L'ancienne interface, qui utilisait la fonction `signal()`, est à proscrire pour des raisons de portabilité.

La fonction `sigaction()` : `int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)`

Exemple :

```
void TraiteSignal ( int sig ) ;
int main ( int argc , char **argv )
{
   struct sigaction act ;

   act.sa_handler = TraiteSignal ;
   sigemptyset ( &act.sa_mask ) ;
   act.sa_flags = SA_RESTART ;
   if ( sigaction ( SIGUSR1 , &act , NULL ) == -1 )
   {
      perror ( "sigaction" ) ;
      exit ( EXIT_FAILURE ) ;
   }
...
void TraiteSignal ( int sig )
{
   printf ( "Réception du signal numéro %d.\n" , sig ) ;
}
```

La fonction `sigaction()` indique au système comment réagir sur réception d'un signal. Elle prend comme paramètres :

1. Le numéro du signal auquel on souhaite réagir.
2. Un pointeur sur une structure de type `sigaction`. Dans cette structure, trois champs nous intéressent :
   - sa_handler, qui peut être :
     - un pointeur vers la fonction de traitement du signal ;
     - SIG_IGN pour ignorer le signal ;
     - SIG_DFL pour restaurer la réaction par défaut.
   - sa_flags, qui indique des options liées à la gestion du signal. Étant donné l'architecture du noyau UNIX, un appel système interrompu par un signal est toujours avorté et renvoie EINTR au processus appelant. Il faut alors relancer cet appel système. Il est possible de demander au système de redémarrer automatiquement certains appels système interrompus par un signal. La constante SA_RESTART est utilisée à cet effet.
   - sa_mask indique la liste des signaux devant être bloqués pendant l'exécution de la fonction de traitement du signal. On ne veut généralement bloquer aucun signal, c'est pourquoi on initialise sa_mask à zéro au moyen de la fonction `sigemptyset()`.

3. Un pointeur sur une structure de type sigaction, structure qui sera remplie par la fonction selon l'ancienne configuration de traitement du signal. Ceci ne nous intéresse pas ici, d'où l'utilisation d'un pointeur nul.

La valeur renvoyée par sigaction() est :
- 0 si tout s'est bien passé.
- -1 si une erreur est survenue. Dans ce cas l'appel à sigaction() est ignoré par le système.

Ainsi, dans notre exemple, à chaque réception du signal SIGUSR1, le programme sera dérouté vers la fonction `TraiteSignal()`, puis reprendra son exécution comme si de rien n'était. En particulier, les appels système qui auraient été interrompus par le signal seront relancés automatiquement par le système.

8 Autres primitives pouvant servir...

8.1 La fonction `system()`

`int system(char* commande)` Elle permet d'appeler une commande UNIX depuis un programme C. Le résultat de la commande est affichée à l'écran. Exemple : `system("ls")` La commande `system()` n'accepte qu'une seule chaîne de caractères en argument et ne permet donc pas d'introduire plusieurs arguments dynamiques comme le fait la fonction `printf()`. Si tel est le cas, il faut alors la coupler avec la fonction `sprintf()`. Cette dernière permettra de construire à la convenance du programmeur la chaîne finale qui sera utilisée par la fonction `system()`.

8.2 La fonction `popen()`

`FILE * popen(char * commande, char * mode)`
- `FILE *` indique que la fonction `popen()` retourne un pointeur sur une structure de type `FILE`, comme le fait la fonction `fopen()`.
- `commande` représente la chaîne de caractères associée au nom de la commande Unix, éventuellement suivie de ses arguments, exactement comme pour la fonction `system()`.
- mode correspond au mode d'entrée/sortie choisi :
  - "r" lecture
  - "w" ecriture

`popen()` ouvre un pipe (fichier temporaire) avant d'appeler la commande, permettant ainsi au programme appelant de communiquer avec le programme appelé grâce à ce pipe.

Exemple de lecture de données d'une commande restituées sur l'écran :

```
FILE *pp;

if ((pp = popen("ps aux", "r") == NULL) {
perror("popen");
exit(1);
}
while (fgets(buf, sizeof(buf), pp))
fputs(buf, stdout);
```

```
pclose(pp);
```

Exemple d'écriture de données vers une commande à partir d'une saisie clavier :

```
FILE *pp;

if ((pp = popen("/usr/ucb/mail", "w") == NULL) {
perror("popen");
exit(1);
}
while (fgets(buf, sizeof(buf), stdin))
fputs(buf, pp);
pclose(pp);
```

La fonction `pclose()` permet de fermer le pipe, avec en argument le pointeur de fichier rendu initialement par `popen()`.